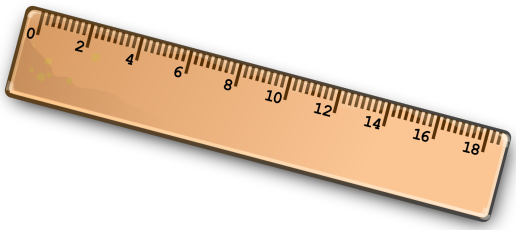
Олимпиадное программирование

Пособие по линейным алгоритмам



Барбулев Илья Денисович

Октябрь 2024

Содержание

1 Предисловие

1.1 Общая информация

Это пособие является индивидуальной выпускной работой (ИВР) Барбулева Ильи Денисовича, ученика 11И3 класса Лицея НИУ ВШЭ.

Оно создано специально для тех, кто только начинает свой путь в олимпиадном программировании и/или имеет рейтинг на Codeforces **менее** 1200.

Автор старался сделать материал доступным и понятным, без использования сложных математических определений и формул. Цель данного пособия – не нагрузить читателя сложными формулировками, а помочь разобраться и углубиться в тему. Строгие доказательства, точные определения и подробные описания можно найти в открытом доступе в интернете.

Если вы обнаружили ошибку / неточность / опечатку, то со мной можно связаться через телеграмм: @llwet4try00.

1.2 Решение задач

Задачи доступны для решения на платформе Codeforces (необходимо вступить в группу): https://codeforces.com/group/Yi3KzYv5u1/contests/

Рекомендую уделить время и попытаться самостоятельно решить задачи после изучения соответствующей темы.

! В начале каждого кода напишите эти 2 строчки:

```
cin.tie(NULL);
ios_base::sync_with_stdio(0);
```

Они значительно ускорят ввод и вывод чисел, что помогает во многих задачах справляться с TL. Также не стоит использовать std::endl, если не требуется выводить данные «онлайн», так как с переводом строки он сбрасывает буфер, что замедляет программу. Вывод с '\n' работает гораздо быстрее.

2 Основные понятия об асимптотике, времени работы программ

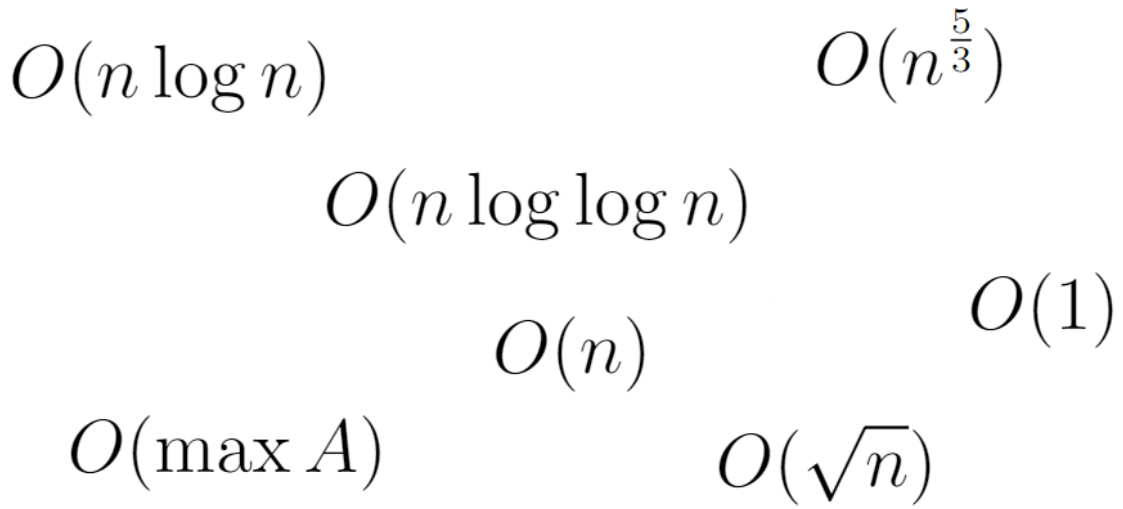
2.1 Введение

В олимпиадном программировании понимание эффективности алгоритмов играет ключевую роль. Одним из основных инструментов для анализа эффективности является концепция **асимптотической сложности** (чаще всего называют просто *асимптотикой*). Этот подход позволяет оценивать время выполнения программы и использование ресурсов в зависимости от размера входных данных.

Научившись определять сложность алгоритмов, мы сможем с легкостью определять и эффективность, сравнивать их между собой и заранее оценивать поведение алгоритма при изменении входных данных.

Запуск алгоритма и замер времени работы в реальных условиях **не может** дать точного представления о поведении и времени работы алгоритма. К тому же результаты измерений могут сильно зависеть от модели процессора, настроек устройства, поэтому перейдем к такому понятию в асимптотике, как *О-большое*.

2.2 О-нотация



O-нотация используется для оценки сверху времени работы алгоритма. Она показывает, как быстро увеличивается время выполнения алгоритма с ростом объема входных данных.

Запись $O(x)$ указывает, что это примерная оценка, не учитывающая времени выполнения отдельных конкретных операций (при условии, что каждая операция выполняется за **фиксированное** время, **не зависящее** от входных данных).

Рассмотрим примеры использования *O-нотации* для оценки сложности алгоритмов:

- ```
  int solve (int a, int b) {
      return a + b;
  }
  ```

  Какие мы бы числа не ввели, программа сделает 2 действия: сложит два числа и выведет ответ. Данный код имеет константную сложность – $O(1)$.

- ```
  int solve (int n, int m) {
      for (int i = 2; i <= m; ++i) {
          if (n % i == 0) {
              return i;
          }
      }
      return -1;
  }
  ```

  В худшем случае, когда наименьший делитель n больше m, цикл сделает все m итераций. Время выполнения алгоритма пропорционально m, его сложность линейная – $O(m)$.

- ```
  void solve (vector<int> &a) {
      int n = a.size();
      for (int i = 0; i < n; ++i) {
          for (int j = 0; j < n; ++j) {
              cout << a[i] << ' ' << a[j] << '\n';
          }
      }
  }
  ```

  На каждой итерации внешнего цикла i цикл j сделает n итераций. Итого сложность является квадратичной – $O(n^2)$.

- ```
  void solve (int n) {
      for (int i = 1; i <= n; i <<= 1) {
          cout << ((n & i) & 1) << ' ';
      }
  }
  ```

  Максимальная степень двойки, не превышающее число n – $\lfloor \log_2 n \rfloor$. Поэтому количество итераций, не может быть больше чем $\lfloor \log_2 n \rfloor$, а значит и сложность программы логарифмическая – $O(\log n)$.

2.3 Определение сложности алгоритма

Давайте определим асимптотику на примере:

Дан массив длины n. Требуется посчитать и вывести сумму сумм всех подотрезков массива.

Рассмотрим решение этой задачи на языке C++:

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; ++i) cin >> a[i];
    long long answer = 0;
    for (int l = 0; l < n; ++l) {
        long long current = 0;
        for (int r = l; r < n; ++r) {
            current += a[r];
            answer += current;
        }
    }
    cout << answer << '\n';
    return 0;
}
```

1) В данном коде присутствуют переменные n – длина массива, a – сам массив. Из них, относящиеся к входным данным является переменная n.

2) С помощью двух циклов for считаем сумму сумм всех отрезков: внешний отвечает за левую границу отрезка, внутренний – за правую. На i-ой итерации внешнего цикла внутренний сделает $(n - i)$ действий:

$$n + (n-1) + \ldots + (n - (n-1)) = n^2 - (1 + 2 + \ldots + (n-1)) =$$

$$= n^2 - \frac{n(n-1)}{2} = \frac{n(n+1)}{2} = \frac{n^2+n}{2}.$$

3) Самое быстрорастущее слагаемое в асимптотике – n^2. Оставляем только его.

4) Коэффициент $\frac{1}{2}$ убирается.

5) Итоговая асимптотика – $O(n^2)$.

2.4 Некоторые свойства асимптотики

- $O(2024) = O(1)$
- $O(\frac{n}{3}) = O(n)$
- $O(n^2 + n\sqrt{n}) = O(n^2 + n^{\frac{3}{2}}) = O(n^2)$
- $O(\frac{n(n-1)}{2}) = O(n^2 - n) = O(n^2)$
- $O(n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + ...) = O(n(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ...)) = O(n \log n)$
- $O(\log_a n) = O(\log n)$

Последнее выполняется, так как логарифмы с разным основанием отличаются друг от друга в **константу** раз.

2.5 Асимптотика и время

Иногда, узнав асимптотическую сложность, непонятно «зайдет» ли по времени алгоритм. Чтобы избежать таких ситуаций, можно воспользоваться трюком.

$$O(n\sqrt{n}) \overset{n \leq 10^5}{\Rightarrow} 10^5\sqrt{10^5} \approx 3 * 10^7$$

Рис. 2: Пример для $O(n\sqrt{n})$ при $n \leq 10^5$

Возьмите максимальные числовые значения переменных из ограничений задачи и подставьте их в асимптотику. Вычислите и теперь поделите на величину, равную кол-ву операций в секунду используемого языка программирования.

Для C++ можно взять 10^8, Для Java 10^7, Для Python 10^6. Получится примерное (довольно грубое) время выполнения в секундах.

2.6 Заключение

Понимание и умение определять асимптотическую сложность алгоритмов являются ключевыми навыками в олимпиадном программировании. Очень важно на данном этапе уметь работать с асимптотикой алгоритмов, чтобы успешно усвоить материал в дальнейших главах.

3 Префиксные суммы

Префиксные суммы – мощный инструмент в олимпиадном программировании, особенно полезный при решении задач, связанных с последовательными суммами элементов массива. Сама идея вычислять что-либо на префиксах и на отрезках часто встречается в задачах. В этой главе мы рассмотрим понятие префиксных сумм, их вычисление и применение в задачах.

3.1 Определение

Префиксные суммы (какого-то массива, допустим массива a длины n) – массив длины $(n + 1)$ (пусть это будет массив p), который на позиции i $(0 \le i \le n)$ содержит сумму всех чисел массива a от 0 до i не включительно – $[0; i)$.

- $p[0] = 0$
- $p[1] = a[0]$
- $p[2] = a[0] + a[1]$
- ...
- $p[i] = a[0] + a[1] + ... + a[i - 1] = \sum_{j=0}^{i-1} a[j]$

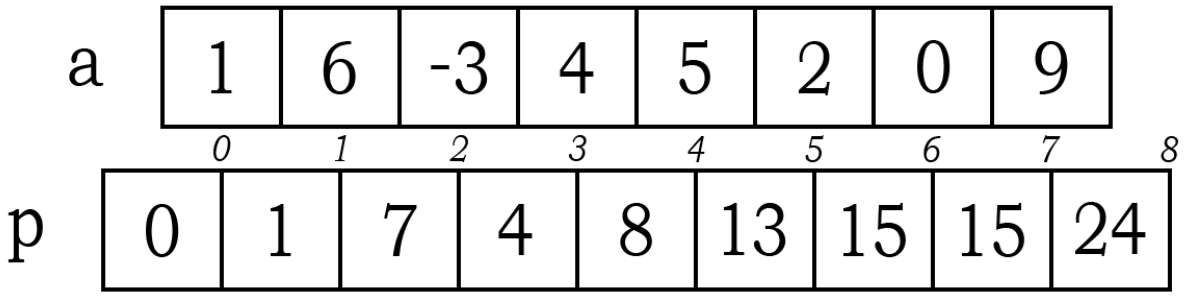


Рис. 3: Массив преф. сумм p для массива a

Иногда префиксные суммы определяют по-другому, храня в i-ой позиции сумму элементов от 0 до i включительно. Такой способ имеет право на существование, но на практике неудобен, но об этом позже.

Построение за $O(n^2)$ очевидно, но давайте посчитаем преф. суммы быстрее.

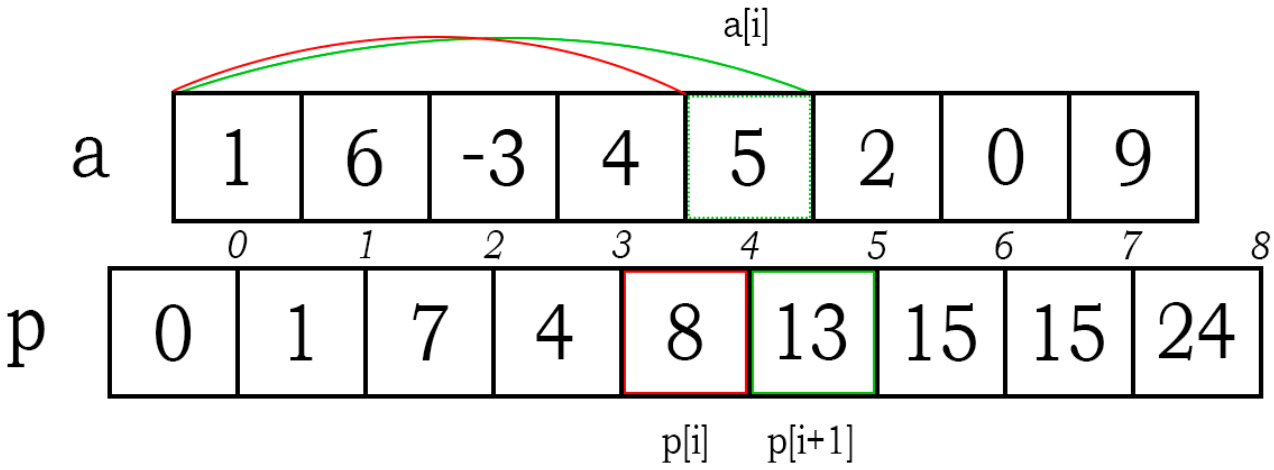


Рис. 4: Пересчёт $p[i+1]$ через $p[i]$

Пусть при переходе к $(i+1)$-ому элементу, у нас все значения от 0 до i посчитаны корректно. Заметим, что сумма элементов на отрезке полуинтервале $[0;i+1)$ можно представить как сумму на $[0;i)+a[i]$. А сумма на $[0;i)$ хранится в $p[i]$. Тоесть мы легко можем пересчитать $p[i+1]$ через $p[i]+a[i]$. Итого мы сможем построить префиксные суммы за $O(n)$, корректно посчитав значение $p[0]$ (сумма на $[0;0)=0$).

Код:

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    int n;              //Размер массива
    vector<int> a(n);   //Сам массив

    vector<long long> p(n+1); //Обратите внимание на тип данных

    for (int i = 0; i < n; ++i) {
        p[i+1] = p[i] + a[i];
    }
    return 0;
}
```

3.2 Применение (1D)

Давайте сразу же решим самую базовую задачу: https://sort-me.org/tasks/10.

Решение: считаем массив a в 0-индексации. Построим префиксные суммы. Теперь будем отвечать на запросы суммы на отрезке. Запрос представляет пару чисел l, r $(1 \le l \le r \le n)$. Так как у нас числа в массиве идут с 0 , отрезок $[l; r]$ можно записать по-другому, как полуинтервал $[l-1; r)$. Пользуясь определением префиксных сумм, ответ на запрос равен $p[r] - p[l-1]$.

Код: https://pastebin.com/KCTaNLmn

! Если бы мы определили префиксные суммы на отрезках, а не на полуинтервалах, то формула суммы на отрезке получилась бы менее лаконичной, ведь нам дополнительно пришлось бы обрабатывать случай суммы на отрезках, левая граница которых равна 0. В данном случае элемента на позиции $(0-1) = -1$ не существует, и вычитать соответственно ничего не придется:

Код (преф. суммы на отрезках): https://pastebin.com/nd9CdV2n

3.3 Многомерные префиксные суммы

Мы уже научились строить префиксные суммы на массиве (одномерные), но они могут быть применены и в бо̀льших размерностях. Например, определим и посчитаем двумерные префиксные суммы:

Пусть у нас есть двумерный массив $a[n][m]$ (n строк и m столбцов). Префиксные суммы (двумерные) массива a – массив $p[n+1][m+1]$, который в i-ой строке j-ом столбце $(0 \le i \le n, 0 \le j \le m)$ содержит сумму всех чисел в прямоугольнике, правый нижний угол которого находится после $(i-1)$-ой строки $(j-1)$-ого столбца.

- $p[i][j] = a[0][0] + a[0][1] + \ldots + a[0][j-1] + a[1][0] + a[1][1] + \ldots + a[1][j-1] + \ldots + a[i-1][j-1] = \sum_{x=0}^{i-1} \sum_{y=0}^{j-1} a[x][y]$

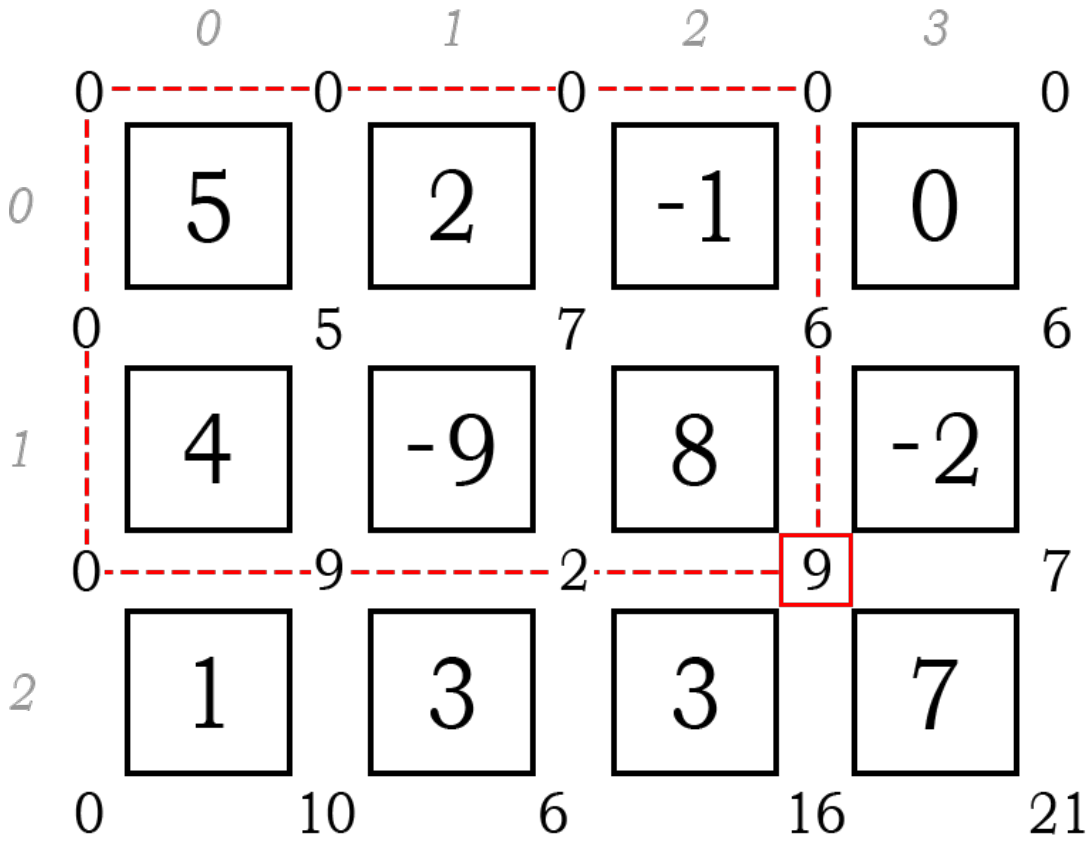


Рис. 5: $p[i][j]$ находится между j-ым и $j+1$-ым столбцом между i-ой и $i+1$-ой строкой

На рисунке в виде квадратов изображены элементы двумерного массива a, и между ними значения массива двумерных префиксных сумм.

Обратите внимание, здесь мы также пользуемся полуинтервалами, чтобы достичь более лаконичную форму. Конечно же, определение с помощью отрезков не запрещено, но количество крайних случаев слишком быстро увеличивается с ростом размерности, что на практике очень неудобно.

Кажется, что двумерные префиксные суммы слишком сложно посчитать за линейное время, ведь если сделать похожий пересчет, как в одномерном случае, то мы посчитаем какие-то элементы несколько раз. Но операция сложения – обратимая, так что давайте какие-то избыточные слагаемые вычтем из суммы, тем самым добьемся корректности.

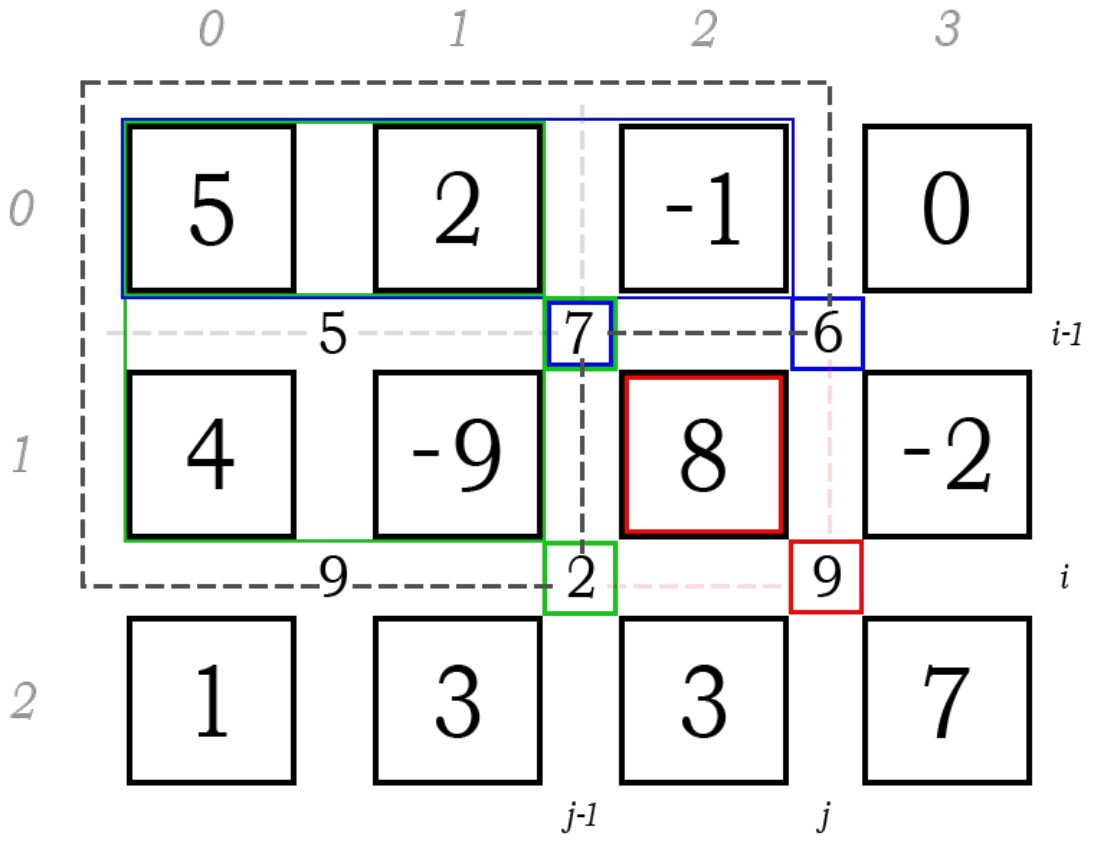


Рис. 6: Пересчет двумерных преф. сумм

Будем считать преф. суммы слева направо, сверху вниз:

Пусть, при переходе к клетке (i, j) все значения до нее были посчитаны корректно. Нам известно значение $a[i][j]$. Осталось узнать сумму оставшейся части (см. рисунок). Она разбивается на прямоугольные части. Верхняя часть – сумма чисел на прямоугольнике от $(0, 0)$ до $(i - 1, j)$. По нашему предположению, сумма чисел в ней равна $p[i - 1][j]$. Левая часть - сумма чисел на прямоугольнике от $(0, 0)$ до $(i, j - 1)$. Сумма равна $p[i][j - 1]$. Но числа, лежащие в пересечении этих двух частей вошли в сумму 2 раза. Пересечение находится в точке $(i - 1, j - 1)$. Нам необходимо вычесть сумму чисел, то есть вычесть $p[i - 1][j - 1]$. Получается $p[i][j] = p[i - 1][j] + p[i][j - 1] - p[i - 1][j - 1] + a[i - 1][j - 1]$.

Код:

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main(){
    int n, m;                                       //Размеры массива
    vector<vector<int> > a(n, vector<int>(m));  //Сам массив

    //Обратите внимание на тип данных
    vector<vector<long long> > p(n + 1, vector<long long>(m + 1));
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            p[i][j] = p[i-1][j] + p[i][j-1] - p[i-1][j-1] + a[i-1][j-1];
        }
    }
    return 0;
}
```

3.4 Применение (2D)

Давайте решим простую задачу: https://codeforces.com/gym/105102/problem/4.

Решение: считаем массив $a[n][m]$ в 0-индексации и построим префиксные суммы. По условию, нам нужно посчитать максимальную сумму чисел в подпрямоугольнике квадратной формы. Переберем размер стороны квадрата, пусть она равна k ($k \leq min(n, m)$). Теперь рассмотрим все квадраты фиксированной длины. Их правый нижний угол не может быть выше строки k, и левее столбца k. Чтобы найти сумму, воспользуемся «трюком» как в пересчете двумерных преф. сумм.

Код: https://pastebin.com/uewi4K4Y

Определение, построение и применение n-мерных префиксных сумм ($n > 2$) автор оставляет читателю в качестве упражнения.

3.5 Другие функции

На **префиксах / суффиксах** массива можно строить много разных функций: побитовое исключающее ИЛИ (XOR), минимум/максимум, НОД, однако не все они могут быть применены чтобы также легко вычислить значение на подотрезке. Чтобы получить значение суммы на подотрезке, мы пользовались обратимостью операции сложения. Однако с максимумом, минимумом и похожими такое не сработает.

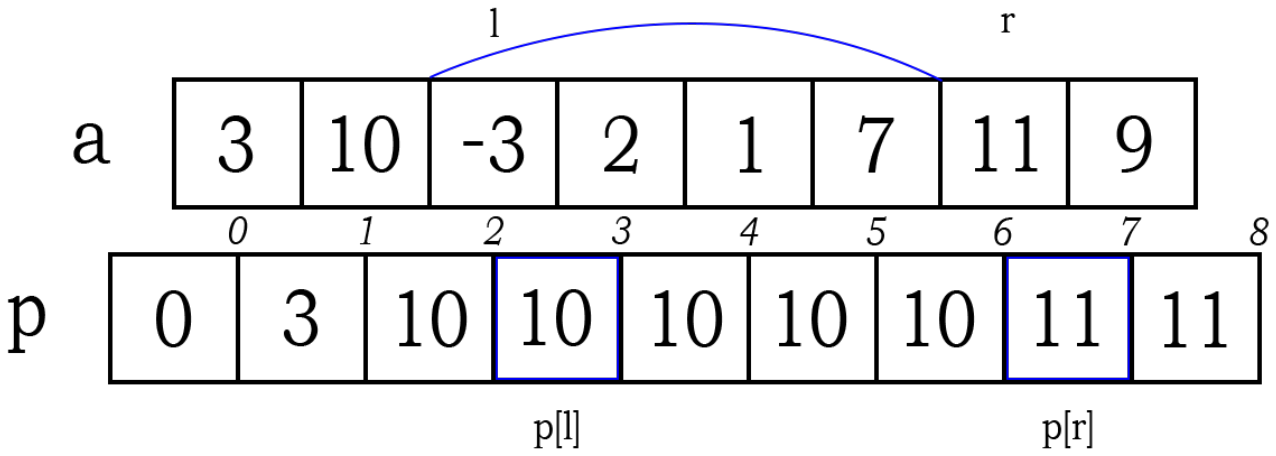


Рис. 7: Массив p префиксных максимумов массива a

Мы видим что на двух концах отрезка (полуинтервала) значения префиксного максимума равны 10, однако само число 10 находится вне отрезка.

С побитовым исключающим ИЛИ все еще проще: эта операция обратима сама себе. Так что значение XOR на подотрезке это сам XOR двух концов полуинтервала $(\bigoplus_{i=l-1}^{r-1} a[i]) = (p[r] \oplus p[l-1])$.

3.6 Практика

Задачи на префиксные суммы и на последующие темы можно прорешать в специально подготовленных контестах в группе пособия на Codeforces. Чтобы отсылать решения, необходимо вступить в группу.

Дополнительные задачи:

- https://leetcode.com/problems/number-of-sub-arrays-with-odd-sum/description/

4 Разностный массив

Давайте рассмотрим еще один инструмент, часто использующийся с префиксными суммами, — разностный массив. Переход к разностному массиву – очень полезный трюк, в некоторых типах задач позволяет существенно улучшить асимптотику, оптимизируя $O(n)$ или $O(\log n)$ до $O(1)$.

4.1 Определение

Разностный массив (какого-то массива, допустим массива a длины n) – массив длины n (пусть это будет массив d), i-й ($0 < i < n$) элемент которого равен разности a[i] и a[i-1] элемента.

- $d[0] = a[0]$
- $d[1] = a[1] - a[0]$
- $d[2] = a[2] - a[1]$
- ...
- $d[i] = a[i] - a[i-1]$

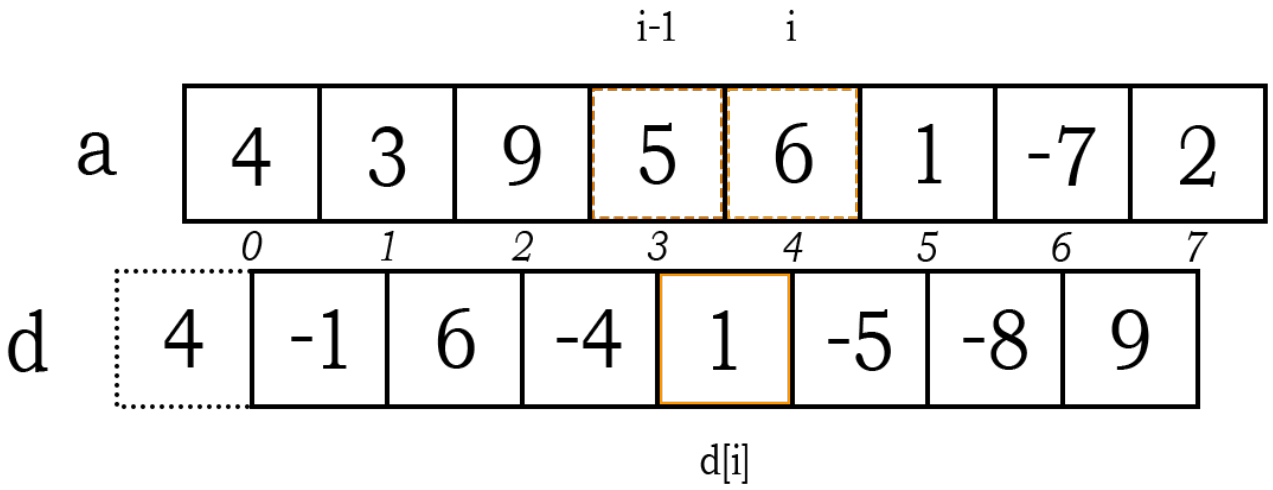


Рис. 8: Разностный массив *d* массива *a*

Фиктивный $a[0]$ добавим в начало, это не повлияет на корректность построения, а наоборот, позволит удобнее работать с массивом.

Код:

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main(){
    int n;              // Размер массива
    vector<int> a(n);   // Сам массив a
    vector<int> d(n);   // Разностный массив d
    for (int i = 0; i < n; ++i) {
        d[i] = a[i];
        if (i) d[i] -= a[i-1];
    }
    return 0;
}
```

Заметим, что если мы построим разностный массив на массиве префиксных сумм массива a, то мы получим исходный массив a. Очевидно, работает и в обратную сторону: если построить массив префиксных сумм по разностному массиву, то также получим исходный массив. В этом несложно убедиться, расписав слагаемые на бумажке: $d[0] + d[1] + d[2] = a[0] + (a[1] - a[0]) + (a[2] - a[1]) = a[2]$.

4.2 Применение

Разберем базовую учебную задачу на разностный массив:

> Дан массив a длины n. Также дано q запросов о прибавлении числа x на отрезке $[l; r]$. После выполнения запросов требуется вывести получившийся массив.

Конечно, если вы знаете структуры данных, задачу можно решить за $O(q\sqrt{n})$ или $O(q \log n)$. Но заметим, что знать конкретные значения элементов массива нам нужно только после всех запросов, поэтому можно обойтись без структур данных, применив такой трюк:

Построим разностный массив d по массиву a. Теперь посмотрим на запрос, а именно что меняется в разностном массиве после прибавления числа на ~~отрезке~~ полуинтервале (в 0-индексации).

После применения разности соседних элементов внутри отрезка не поменялись $(a[i]+x-(a[i-1]+x) = a[i]-a[i-1])$. Поменялись разности только на концах отрезка: в $d[l]$ она стала на x больше, а в $d[r]$ (если такой элемент существует) на x меньше. $d[l] = (a[l+1]+x)-a[l]$ и $d[r] = a[r]-(a[r-1]+x)$.

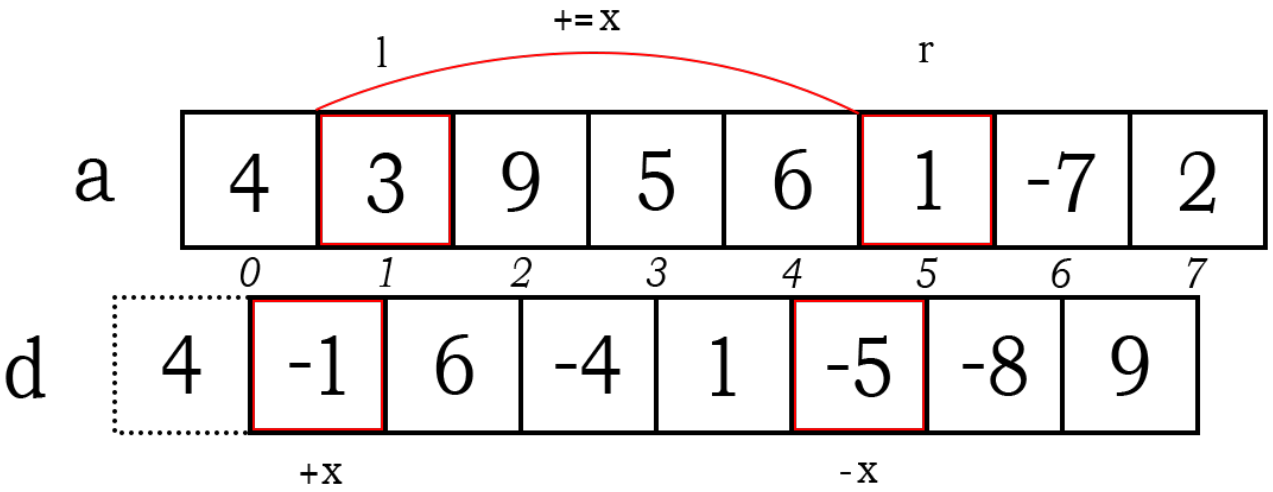


Рис. 9: Изменение разностного массива после прибавления на отрезке x

Получается, мы можем за $O(1)$ обработать прибавление на отрезке, поменяв 2 значения в точке в разностном массиве. В конце, после запросов, узнаем получившийся массив. Итоговая асимптотика $O(n+q)$.

Код: https://pastebin.com/6JDmc0jq

Разберем вторую, не менее интересную задачу:

Дан массив a длины n. Также дано q запросов о прибавлении арифметической прогрессии на отрезке $[l;r]$. Более формально: прибавить 1 элементу на отрезке 1, второму 2, .., i-ому – i. После выполнения запросов требуется вывести получившийся массив

Заметим, что если прибавляемая прогрессия имеет произвольное начальное значение и шаг (A_1 и d соответственно), то запрос разбивается на две части: прибавление прогрессии $0, d, 2d, \ldots, (r-l)d$ и прибавление A_1 на отрезке.

По аналогии с предыдущей задачей выясним, как меняется разностный

массив после прибавления прогрессии на каком-либо отрезке.

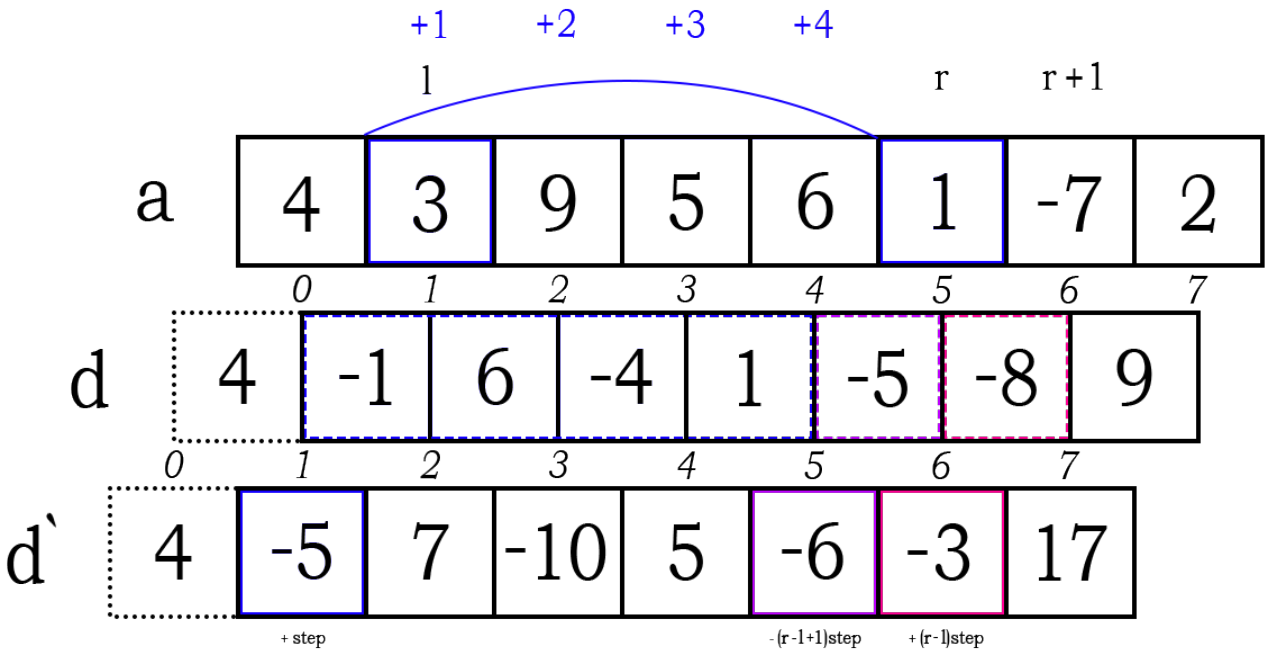


Рис. 10: Изменение разностного массива после прибавления прогрессии на отрезке

Рассмотрим чуть более общее решение с произвольным шагом $step$ (в текущей задаче $step = 1$). Разности соседних элементов внутри отрезка изменились на $step$. Разность l-ого и $(l+1)$-ого (в полуинтервалах) изменилась также на $step$. Разность r-ого и $(r-1)$-ого изменилась на $-(r-l) * step$ (в полуинтервалах). Получается, мы свели задачу к прибавлению на отрезке уже на разностном массиве. Такую мы научились решать, поэтому построим на первом разностном массиве второй. Асимптотика: $O(n+q)$.

Код: https://pastebin.com/yS4WpWaq

4.3 Практика

Дополнительные задачи:

- https://leetcode.com/problems/find-the-power-of-k-size-subarrays-ii/
- https://leetcode.com/problems/special-array-ii/description/

5 Скользящее окно

Скользящее окно – довольно известный прием в олимпиадном программировании, который используется если требуется что-то посчитать на подотрезках **фиксированной** длины. В интернете можно найти целые видео о б этой теме, с объяснением и разбором задач, часто встречающихся на собеседованиях в IT компании. Давайте сразу решим очень простую задачу.

5.1 Применение

I. Дан массив a длины n и число k $(1 \le k \le n)$. Найдите максимальную сумму k подряд идущих элементов в массиве.

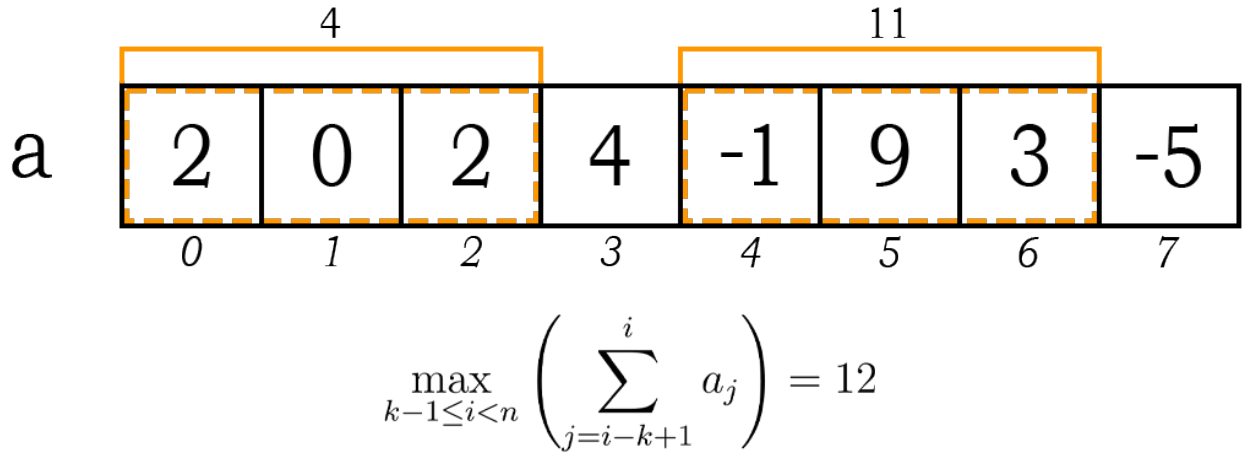


Рис. 11: Максимальная сумма чисел отрезка длины 3 = 12

В голову приходит очевидное решение: зафиксируем правый конец «окна» i, и начиная с левого конца $i - k + 1$ просуммируем числа и обновим ответ. Асимптотика такого решения: $O(nk)$. Это довольно плохая асимптотика, ведь если значение $n \le 10^6$ то при $k = 5 * 10^5$ количество операций будет $\approx 25 * 10^{10}$. Давайте как-то сохраним информацию о рассматриваемых k элементов окна. В данном случае можно просто хранить переменную суммы. При сдвиге окна в сторону нужно удалить последний крайний элемент (он перестанет попадать в окно) и добавить следующий. Получается мы вычитаем из суммы крайний элемент и прибавляем новый. Теперь сумма посчитана корректно, и мы можем обновить ответ. Асимптотика $O(n)$.

Код:

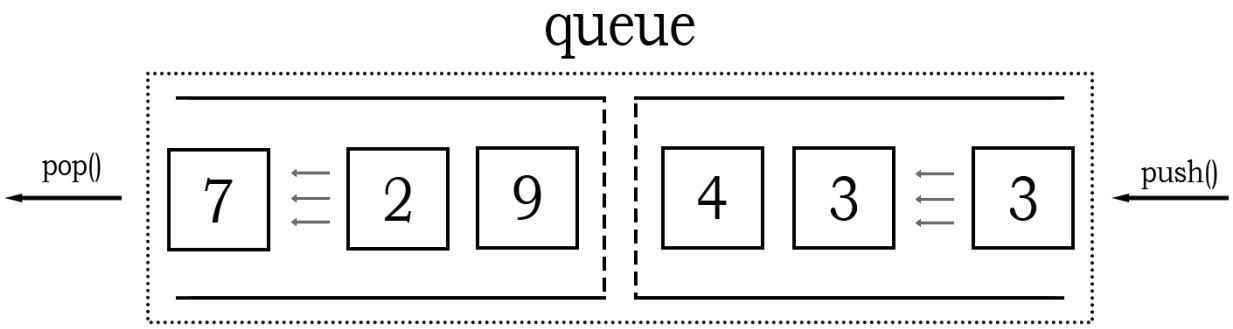
```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    int n, k;               //Размер массива
    vector<int> a(n);    //Сам массив

    long long sum = 0;  //Переменная суммы
    long long ans = -1e18;  //Переменная ответа, изначально -inf
    for (int i = 0; i < k; ++i) {   //Отдельно обрабатываем самое левое окно
        sum += a[i];
    }
    ans = max(ans, sum);     //Обновляем ответ

    for (int i = k; i < n; ++i) {
        sum -= a[i-k];
        sum += a[i];
        ans = max(ans, sum); //Обрабатываем остальные окна
    }
    return 0;
}
```

Также существует другое решение, основанное на префиксных суммах.

5.2 Продвинутые применения

Что делать, если просят посчитать минимум, максимум или другие функции в окне? Не стоит паниковать и писать дерево отрезков или другие сложные структуры данных. В случае с суммой мы хранили информацию в виде одной переменной. Но с минимумом/максимумом такое не получится.

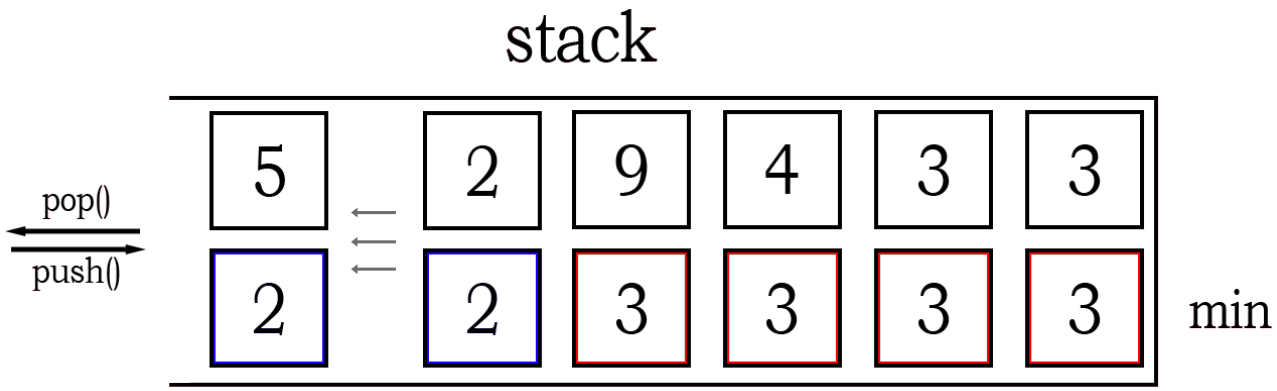
Давайте хранить элементы в структуре данных, в которой можно добавлять элементы в конец и удалять элементы из начала. Для этого идеально подойдет очередь. Но что же делать с пересчетом функции в окне? Если использовать стандартную очередь, то быстрее $O(k)$ узнать ответ не получится. Используя множества или мультимножества, в асимптотике появится лишний логарифм. Чтобы решить эту проблему, напишем свою очередь:



Очередь можно представить в виде двух стеков. При добавлении элемент кладется в правый стек. При удалении убирается из левого стека. Если в момент удаления левый стек пустой, но в правом есть элементы, то просто перекинем их в левый. В таком случае $pop()$ у очереди сработает за $O(m)$, где m равно количеству элементов в правом стеке, но асимптотика останется *амортизированной* $O(1)$, ведь после того, как мы перекинем элементы за $O(m)$, функция $pop()$ продолжит работать за $O(1)$. Чтобы она еще раз сработала за $O(m)$, необходимо удалить как минимум m элементов (все эти удаления будут за чистые $O(1)$), значит каждые m операций у нас будет максимум $O(m)$ удалений из левого стека. Итого $\frac{O(m)}{m} = O(1)$.

Теперь добавим возможность считать функцию, например минимум в очереди. Давайте посчитаем минимум в правом и левом стеке очереди, а при запросе, объединим ответы (это будет минимум из двух минимумов).

Чтобы посчитать минимум на стеке, воспользуемся идеей префиксных сумм: для i-го элемента в стеке будем хранить минимум среди i-го элемента, $(i-1)$-го, . . . , 0-го в стеке.



Чтобы это поддерживать, при добавлении, минимум равен (если стек не пустой) минимуму из последнего элемента в стеке и добавляемого числа. При удалении ничего пересчитывать не придется, нужная актуальная информация лежит «под» удаляемым элементом.

Код:

```
struct min_queue {
    vector<pair<long long,long long> > L, R;

    void push(long long x) {
        if (!R.empty())
            R.emplace_back(min(x, R.back().first), x);
        else
            R.emplace_back(x, x);
    }
    void pop() {
        if (L.empty()) {
            while(!R.empty()) {
                if (!L.empty()) { //push() для левого стека
                    long long cur = min(R.back().second,L.back().first);
                    L.emplace_back(cur, R.back().second);
                }
                else {
                    L.emplace_back(R.back().second, R.back().second);
                }
                R.pop_back();
            }
        }
        L.pop_back();
    }
    long long get() {
        long long F = (long long)1e18, S = (long long)1e18; //нейтральный элемент inf
        if (!L.empty()) F = L.back().first;
        if (!R.empty()) S = R.back().first;
        return min(F, S);
    }
};
```

Реализация в виде структуры очень удобная: если в задаче попросят посчитать 2 или более функции, то вместо копипасты с большим количеством замен можно определить другую очередь и добавить конструктор с определением нужной функции.

5.3 Практика

Дополнительные задачи:

- https://atcoder.jp/contests/abc361/tasks/abc361_c
- https://atcoder.jp/contests/abc352/tasks/abc352_d
- https://sort-me.org/tasks/2368
- https://leetcode.com/problems/find-the-power-of-k-size-subarrays-ii/

6 Два указателя

Техника двух указателей (или метод двух указателей) позволяет решать широкий класс задач за линейное время. В некоторых случаях эта техника может быть обобщена на использование бо̀льшего количества указателей. В этой главе мы рассмотрим метод двух указателей на несложных примерах, найдем условия его применения и разберём несколько задач.

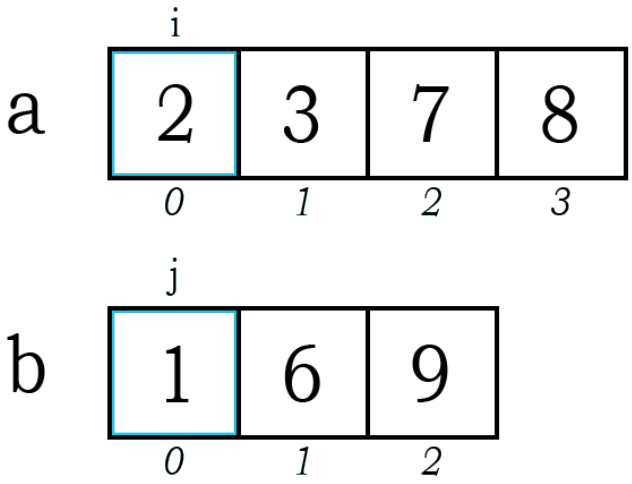
6.1 Примеры задач

Посмотрим, в чем заключается метод на нескольких учебных задачах:

I. Есть 2 отсортированных массива: a длины n и b длины m. Требуется объединить их в один отсортированный массив c длиной $n + m$.

Данная задача очень легко решается за $O((n+m)\log(n+m))$. Объединим любым способом элементы массива a и b и отсортируем. Но данную задачу можно решить за линейное время $O(n + m)$.

Посмотрим на пример:



Так как оба массива отсортированы, первый элемент массива c равен либо первому из a, либо первому из b. В данном случае, $a[0] > b[0]$, а значит 1-ый элемент массива b будет стоять на 1-ой позиции массива c.

Удалим первый элемент из массива b и перейдем ко второму элементу массива c – он равен либо первому из массива a, либо второму из массива b. Снова выберем наименьший из них, удалим и продолжим процесс, учитывая уже использованные элементы.

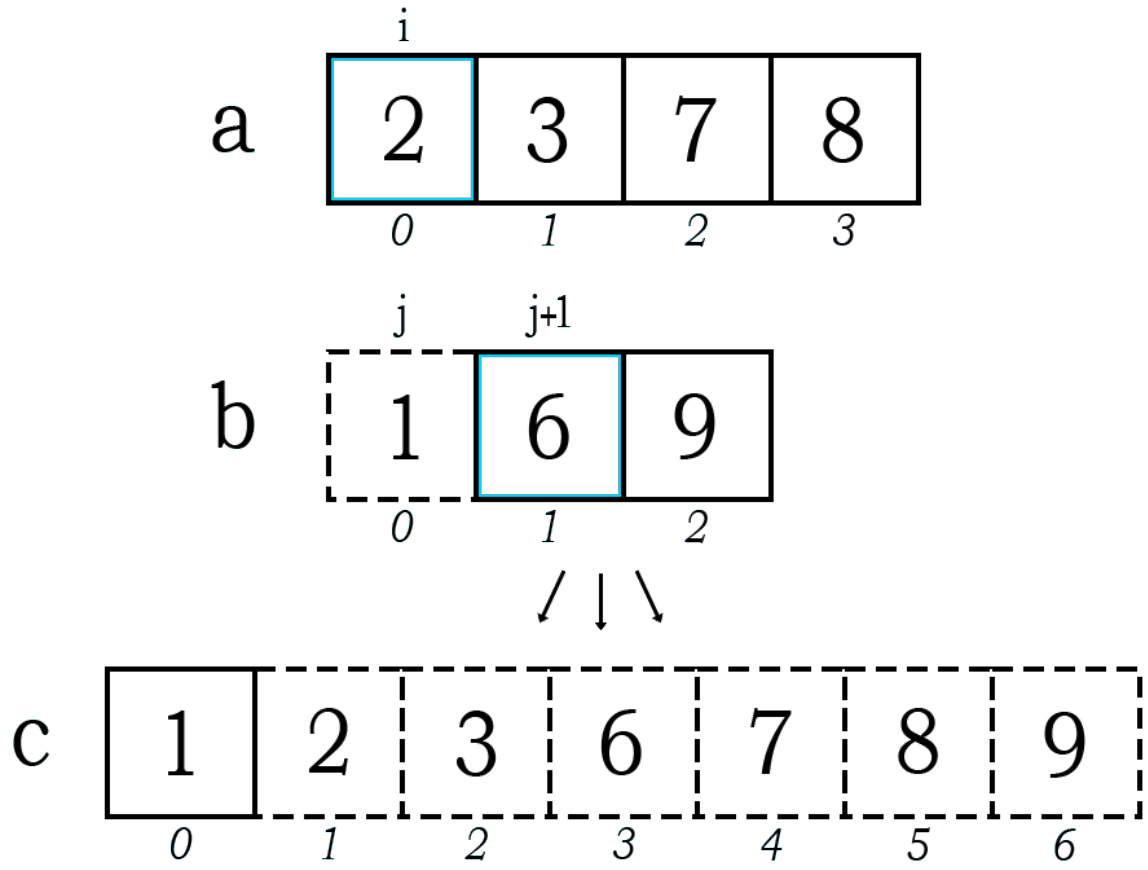


Рис. 15: Слияние двух массивов a и b в один массив c

Чтобы реализовать данный алгоритм, не удаляя явно элементы, заведем два указателя: они будут указывать на первые элементы двух массивов a и b. При удалении элемента, соответствующий указатель сдвигается на 1 вправо. Если указатель вышел за границу массива, то соответствующий массив пустой, поэтому нужно аккуратно обработать этот случай.

Код:

```
void merge(vector<int> &a, vector<int> &b, vector<int> &c) {
    int i = 0, j = 0;
    while (i < n || j < m) {
        int A = (i < n ? a[i] : (int)2e9); //нейтральный элемент inf
        int B = (j < m ? b[j] : (int)2e9);
        if (A < B)
            c.push_back(a[i++]);
        else
            c.push_back(b[j++]);
    }
}
```

Данный алгоритм сделает ровно $n + m$ действий: Каждую итерацию мы сдвигаем ровно один из указателей ровно на 1 вправо. Каждый указатель не может быть сдвинут больше чем на размер соответствующего массива (в данном случае n и m), а значит суммарное количество действий составит $n + m$.

! В C++ есть встроенная функция объединения двух отсортированных массивов a и b – $merge(a, a+n, b, b+m, c)$. Тем не менее, важно понимать, как работает эта функция.

II. Есть 2 массива a и b длины n и m соответственно. Требуется для каждого i $(0 \le i < n)$ найти наибольшее $b[j]$ $(0 \le j < m)$, такое что $b[j] \le a[i]$, или сообщить, что такого значения нет.

Отсортируем массив a, запомнив для каждого элемента индекс в оригинальном массиве (это нам понадобится для восстановления ответа). Отсортируем массив b. Заведем указатель j для массива b. Изначально $j = -1$. Переберем i. Так как массив a отсортированный, то **свойство** $(b[j] \le a[i])$ **сохраняется** при переходе к $i + 1$ элементу: Когда мы перейдем к $i + 1$ элементу, все элементы от 0 до j все также будут не больше $a[i + 1]$: $(b[j] \le a[i] \le a[i + 1])$. Но возможно, какие-то элементы из b, правее j, тоже подходят. Все такие «новые» подходящие значения попробуем узнать, увеличивая j на 1, пока $(b[j + 1] \le a[i + 1])$ и обновим ответ.

Вторая часть алгоритма работает за $O(n+m)$: граница i увеличивается на 1, начиная с 0 до $(n-1)$. Граница j **всегда** только увеличивается на 1. Она не может увеличиваться больше m раз, поэтому суммарно будет выполнено не более $n+m$ действий. Не смотря на всю красоту и линейность второй части, получаем итоговую сложность $O(max(n,m)\log max(n,m))$ из-за сортировки.

Код:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main(){
    int n, m;
    vector<int> a(n), b(m);
    vector<pair<int, int> > t(n);
    for (int i = 0; i < n; ++i) t[i] = make_pair(a[i], i);
    sort(t.begin(), t.end());
    sort(b.begin(), b.end());
    vector<int> ans(n, -1); //Массив для ответов
    int j = -1; //Указатель
    for (int i = 0; i < n; ++i) {
        while (j + 1 < m && b[j+1] <= t[i].first) ++j;
        if (j != -1) ans[t[i].second] = b[j]; //Если элемент найден
    }
    return 0;
}
```

6.2 Задачи с отрезками

Давайте разберем основные задачи на нахождение количества отрезков или макс. длины отрезка в массиве с заданным свойством. Начнем с такой задачи:

III. Дан массив a состоящий из n неотрицательных чисел и число x. Найти количество подотрезков массива с суммой чисел не больше x.

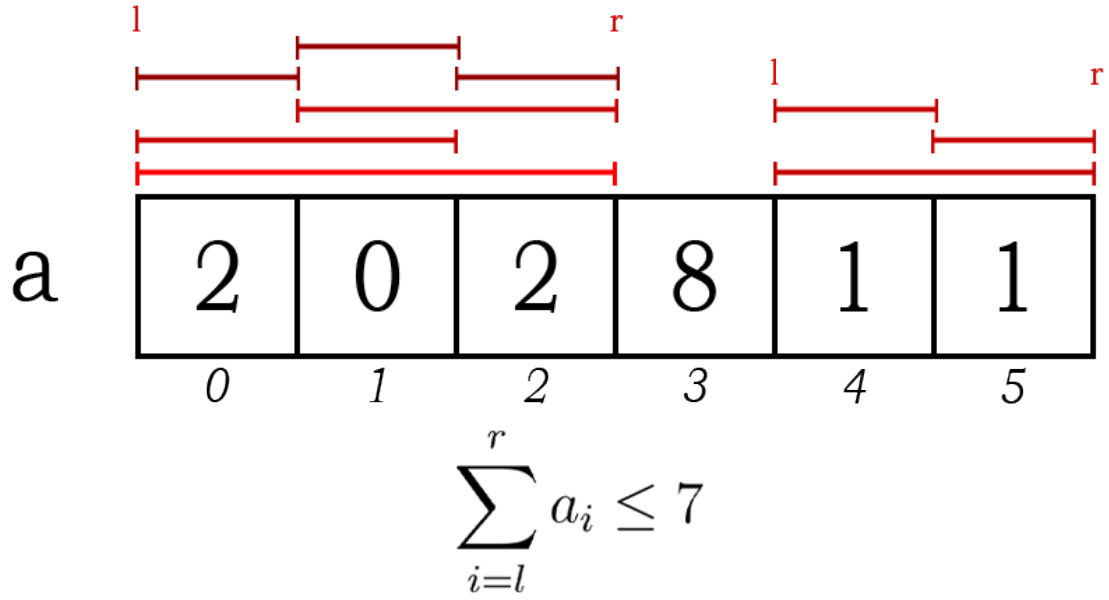


Рис. 16: Подотрезки с суммой не больше 7

Заметим, что числа в массиве неотрицательные, это значит при фиксированной левой границе подотрезка, чем дальше правая граница, тем **не меньше** будет сумма. Давайте для каждой правой границы поддерживать минимальную левую, такую что сумма на отрезке не больше x. По свойству выше, количество отрезков = количеству подходящих правых границ для текущей левой = длине отрезка $(r - l + 1)$.

Код:

```
#include <iostream>
#include <vector>
using namespace std;
signed main(){
    int n;
    long long x;
    vector<int> a(n);
    long long ans = 0, sum = 0;
    int l = 0; //Указатель на левую границу
    for (int r = 0; r < n; ++r){
        sum += a[r]; //Двигаем правую границу
        while (sum > x) sum -= a[l++]; //Сдвигаем левую границу
        ans += max(0, r - l + 1); //Обновляем ответ
    }
    return 0;
}
```

Асимптотика алгоритма – $O(n)$ (по аналогии с задачей II).

IV. Дана бинарная строка s длины n. Найти максимальную длину отрезка, в котором 1 и 0 чередуются.

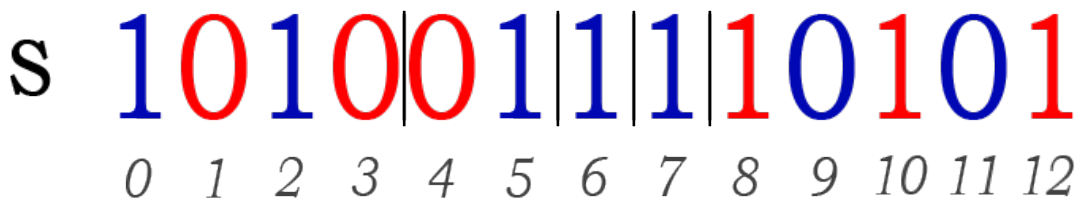


Рис. 17: Разбиение строки s на подотрезки

Давайте выделим все самые длинные подотрезки с чередованием в виде пары (l, r). В отличии от предыдущей задачи, сразу после того как мы найдем такой подотрезок (l, r), нам нет смысла учитывать левые границы l_1, содержащиеся внутри $(l < l_1 \leq r)$. Поэтому продолжим искать ответ правее текущей правой границы, начиная с $r + 1$. Чтобы найти ответ, достаточно вывести максимальное значение $(r - l + 1)$ по всем выделенным (l, r).

Код:

```cpp
#include <iostream>
#include <vector>
using namespace std;
signed main(){
    int n;
    string s;
    int l = 0, r = 0; //Начинаем с начала строки
    int ans = 0;
    while (r < n) {
        while (r + 1 < n && s[r+1] != s[r]) ++r; //Двигаем правую границу
        ans = max(ans, r - l + 1); //Обновляем ответ
        l = ++r; // Более подробно: l = r + 1, r += 1
    }
    return 0;
}
```

Асимптотика – $O(n)$. Мы рассмотрим каждую позицию не более 1 раза.

6.3 Практика

В большинстве задач на тему «2 указателя» применяется конструкция из III задачи (с поиском левой границы для каждой правой). Однако иногда удобнее выделять отрезки и сохранять информацию о них конструкцией из IV задачи.

Советую дополнительно изучить тему и порешать задачи на этом курсе от ITMO Academy: https://codeforces.com/edu/course/2.

Дополнительные задачи:

- https://atcoder.jp/contests/abc326/tasks/abc326_c
- https://sort-me.org/tasks/2367
- https://codeforces.com/gym/104949/problem/2

7 Монотонный стек

Монотонный стек (Стек минимумов / максимумов) – это эффективная структура данных, которая находит широкое применение в задачах, связанных с поиском минимального / максимального элемента в массиве.

7.1 Определение

Определим монотонный стек минимумов (для максимумов определяется аналогично):

Стек минимумов массива a длины n – стек s размера m, в котором во время построения на позиции i в вершине хранится **ближайший** элемент **меньший** чем $a[i]$.

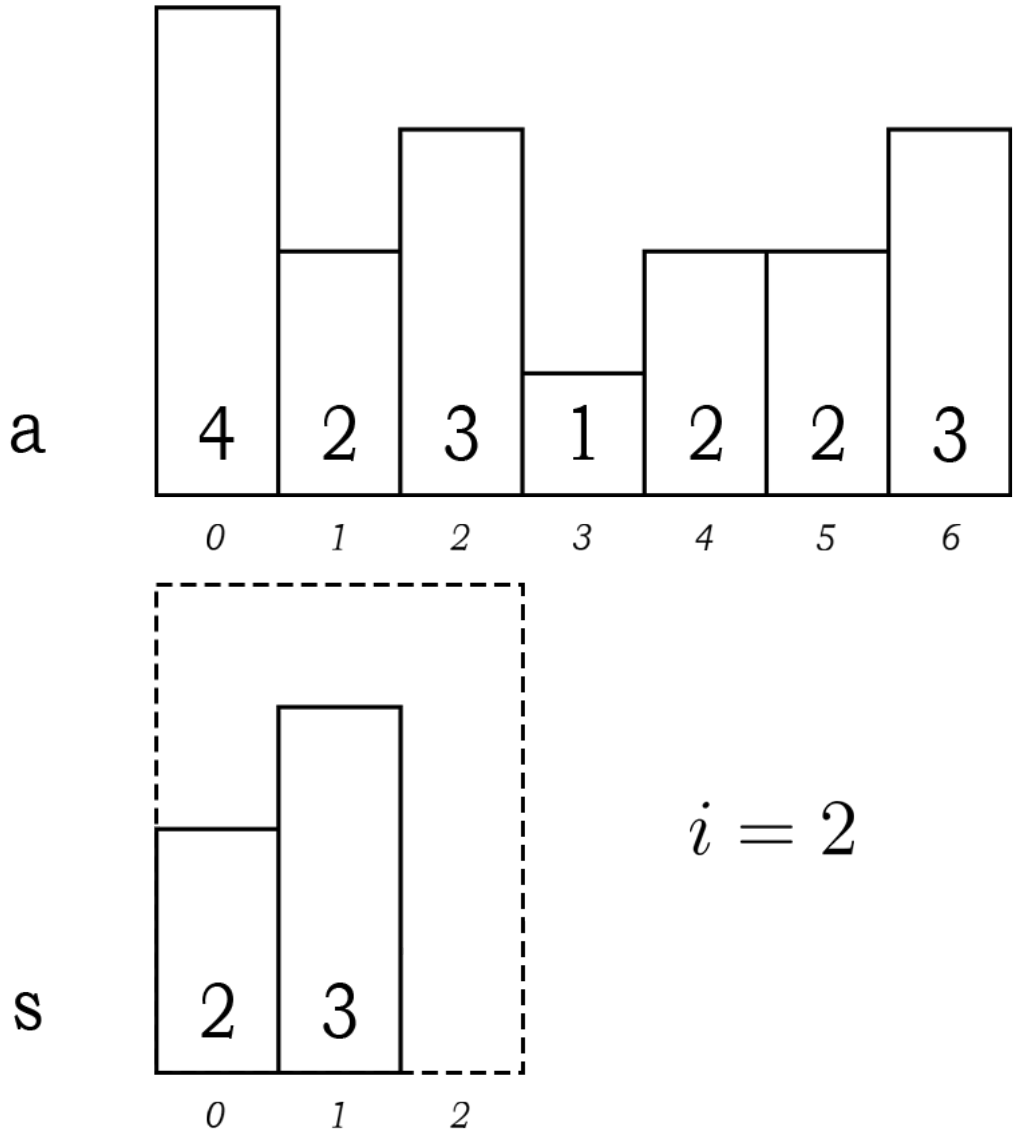


Рис. 18: Стек минимумов s массива a при $i = 2$

Стек минимумов строится последовательно, проходя по всем элементам массива a. Мы удаляем верхний элемент стека, пока стек не пустой и элемент на позиции, равной вершине стека **не меньше** чем текущий $a[i]$. Так мы получим *ближайший меньший элемент* (если стек оказался **пустым**, то для удобности ближайший меньший элемент равен $-\infty$ в идексе (-1). Затем добавим в стек сам $a[i]$.

Код:

```
#include <iostream>
#include <vector>
using namespace std;
signed main(){
    int n;
    vector<int> a(n);
    vector<int> l(n), s;
    for (int i = 0; i < n; ++i) {
        while (!s.empty() && a[s.back()] >= a[i]) {
            s.pop_back(); //Удаляем, пока не найдем подходящий элемент
        }
        if (s.empty()) l[i] = -1;
        else l[i] = s.back();
        s.push_back(i);
    }
    return 0;
}
```

Чтобы находить ближайшие меньшие элементы **справа**, нужно просто поменять порядок внешнего цикла (итерироваться от $n-1$ до 0).

! Стек минимумов (максимумов) можно определять и на **нестрогие** условия, чтобы находить ближайшие элементы не больше (не меньше) текущего. Для этого достаточно изменить знак в условии внутри цикла while.

Сложность построения стека $O(n)$: каждую позицию мы добавим 1 раз и удалим не более 1 раза.

Существует реализация, позволяющая строить массив ближайших меньших элементов без стека:

Код:

```
#include <iostream>
#include <vector>
using namespace std;
signed main(){
    int n;
    vector<int> a(n);
    vector<int> l(n);
    for (int i = 0; i < n; ++i) {
        l[i] = i - 1;
        while (l[i] >= 0 && a[l[i]] >= a[i]) {
            l[i] = l[l[i]];
        }
    }
    return 0;
}
```

Этот способ также работает за $O(n)$, только работа со стеком *неявная*. На практике такой трюк работает немного быстрее из-за более простого обращения к памяти.

7.2 Применение

Давайте решим следующую задачу: https://codeforces.com/contest/547/problem/B.

Нас просят для каждого x $(1 \leq x \leq n)$ найти максимальный минимум на отрезках длины x.

Если зафиксировать подотрезок, найти минимум и обновить ответ для длины, получится решение за $O(n^3)$. Если зафиксировать левую границу и поддерживать минимум, перебирая правую, то асимптотика станет чуть лучше – $O(n^2)$. Но задачу можно решить за $O(n)$:

Давайте будем фиксировать не отрезок и искать минимум, а фиксировать минимум и искать для него отрезки. Посмотрим на i-ый элемент. Какое условие должно соблюдаться у отрезка, чтобы $a[i]$ был минимумом?

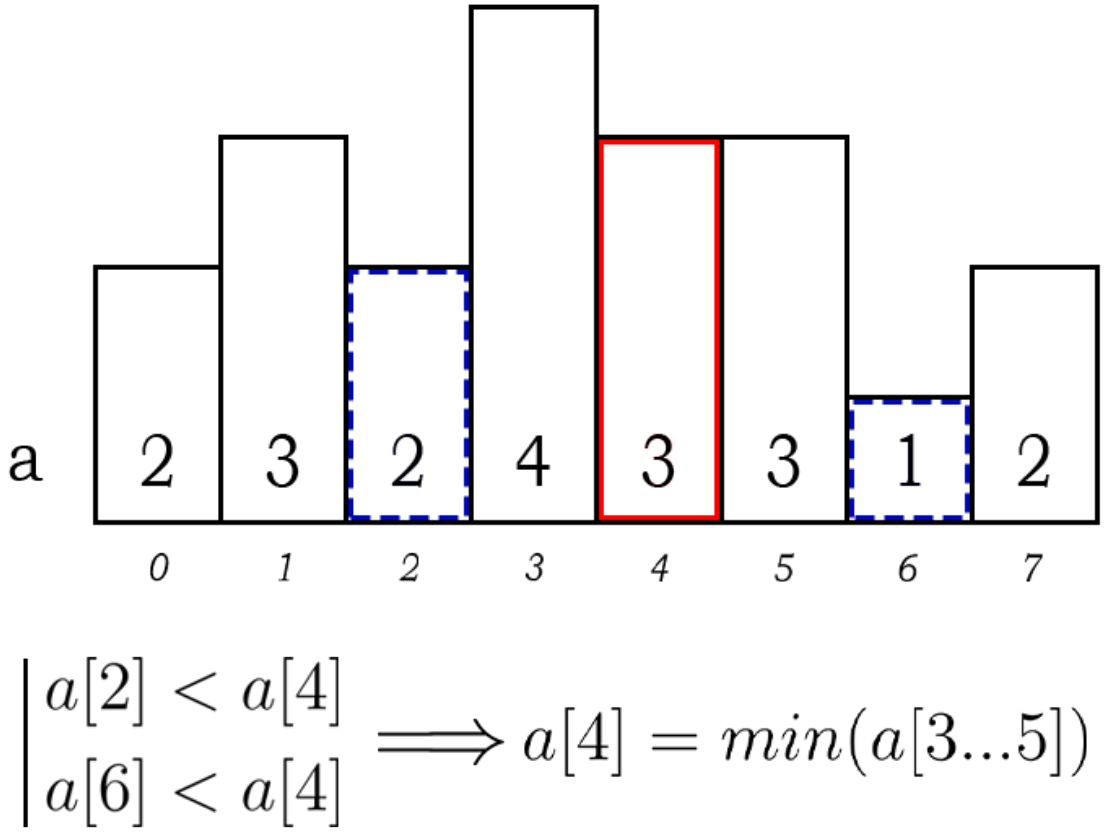
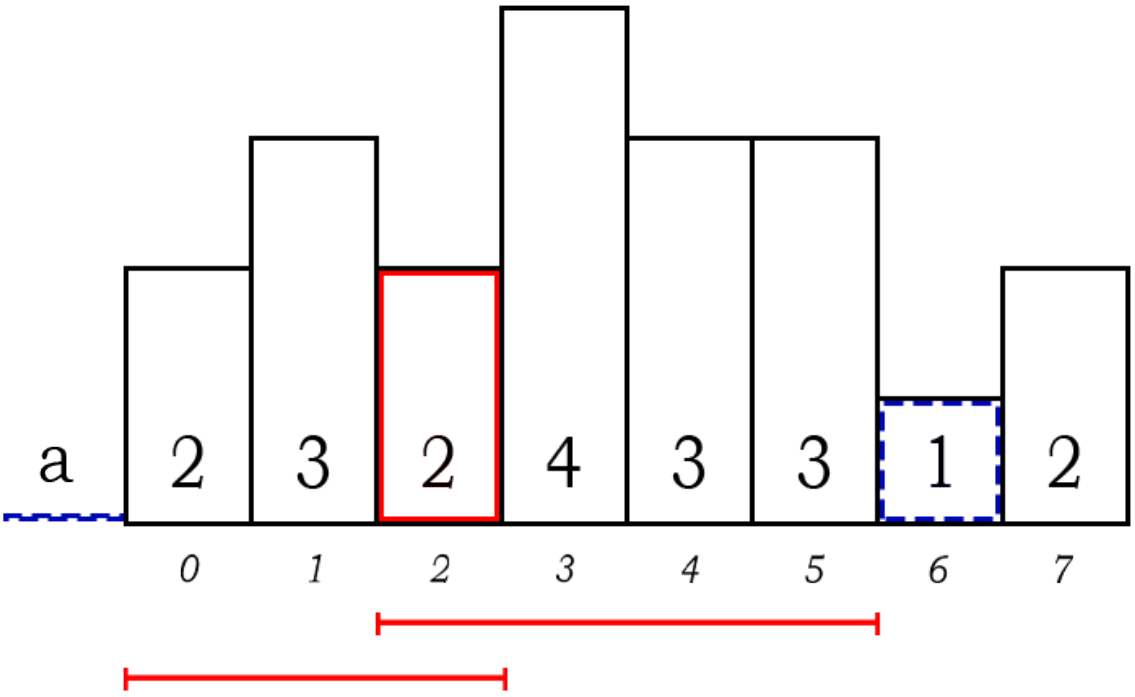


Рис. 19: Ближайшие меньшие $a[4]$ элементы

Очевидно, на отрезке не должно быть элементов **меньших** $a[i]$. То есть, ближайший элемент **справа** от i, меньший $a[i]$ должен быть **правее правой** границы отрезка, аналогично ближайший элемент **слева** от i, меньший a[i] должен быть **левее левой** границы отрезка (если ближайший не попадает в отрезок, то все остальные точно не будут попадать).

Найдем для каждого i $(0 \leq i < n)$ позиции ближайших слева и справа элементов меньших $a[i]$ с помощью стека минимумов и запишем в массивы L и R соответственно. Вернемся к текущему i-ому элементу. Для каких длин он повлияет на ответ? Для длин j, таких что $1 \leq j \leq (R[i] - L[i] - 1)$ обновим ответ $ans[j] = max(ans[j], a[i])$.



$$a[2] = min(a[l_i ... r_i])$$
$$-1 < l_i \leq 2$$
$$2 \leq r_i < 6$$

Рис. 20: Отрезки возможных левых и правых границ при которых минимум равен $a[2]$

Максимальная длина отрезка, на котором $a[i]$ является минимумом, равна $R[i] - L[i] - 1$ ($R[i]$ и $L[i]$ **исключаем**). Существует способ выбрать хотя бы 1 отрезок длины от 1 до максимальной так, чтобы $a[i]$ остался минимумом на нем. Поэтому мы обновим ответ корректно.

Пока что, решение все равно работает за $O(n^2)$, в худшем случае нам придется для каждого $a[i]$ обновить n значений для длин. Но заметим, что с **возрастанием** длины отрезка значение максимального минимума **не возрастает**.

Доказательство:

Пусть m_{k+1} – максимальный минимум на подотрезках длины $(k+1)$. Значит, найдется **хотя бы** 1 подотрезок длины $(k+1)$ с минимумом равным m_{k+1}.

Любой подотрезок длины $k+1$ можно рассмотреть как подотрезок длины k и еще 1 элемент. Минимальное значение на подотрезке длины $k+1$ **не может** быть больше минимального значения на подотрезке длины k, так как добавление нового элемента в массив может либо **сохранить**, либо **уменьшить** минимальное значение. Поэтому выполняется $m_{k+1} \leq m_k$.

Поэтому, обновим ответ для максимальной длины отрезка с минимумом равным $a[i]$, а в конце найдем ответ для остальных длин, пользуясь свойством выше.

Код: https://pastebin.com/CKegm0HG

7.3 Практика

В основном, монотонный стек используется только для определения ближайших меньших / больших элементов для позиций, что помогает в задачах зафиксировать минимум / максимум, отрезок на котором минимум / максимум = min / max и что-то вычислять.

Дополнительные задачи:

- https://atcoder.jp/contests/abc359/tasks/abc359_e
- https://leetcode.com/problems/sum-of-subarray-minimums/description/
- https://sort-me.org/tasks/19

8 Сканирующая прямая

Метод сканирующей прямой (сканлайн) заключается в сведении объектов в задаче к *«событиям»* по каким-то признакам и последующему рассмотрению их по свойствам (чаще всего по *координате* и *времени*).

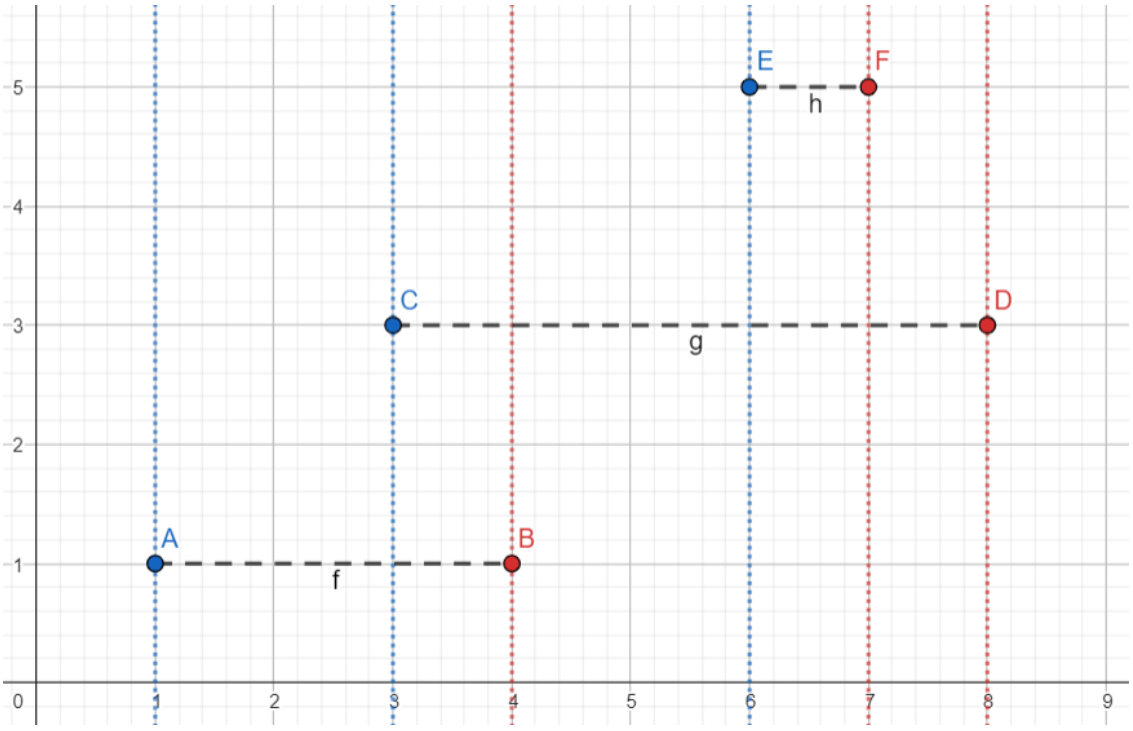
8.1 Примеры задач

Как и с темой «Два указателя», метод сканирующей прямой проще понять, сразу рассмотрев применения в задачах.

I. Дан набор из n ($n \le 10^5$) отрезков $[l_i; r_i]$ на числовой прямой ($0 \le l_i \le r_i \le 10^9$). Найти площадь объединения отрезков.

Если бы координаты были небольшие, то мы могли проставить 1 на точки прямой (сопоставив точку с элементом в массиве), покрытые хотя бы одним отрезком и вывести количество единиц.

Давайте разобьем каждый отрезок на 2 точки: точка левого конца отрезка l_i и правого r_i. Также отметим для каждой точки, начинается ли в ней отрезок или заканчивается.



Теперь, пройдемся по точкам слева-направо (при равенстве координат рассмотрим сначала открывающие отрезок точки, затем закрывающие) и будем поддерживать счетчик *cnt* - количество «открытых» отрезков.

При рассмотрении «открывающей» точки прибавляем 1 к *cnt*, а при «закрывающей» уменьшаем *cnt* на 1. Точка, в которой *cnt* **впервые** стал равен 1 - точка начала отрезка из объединения. Точка, в которой cnt стал равен 0 (**но до этого** $cnt > 0$) - точка конца отрезка из объединения.

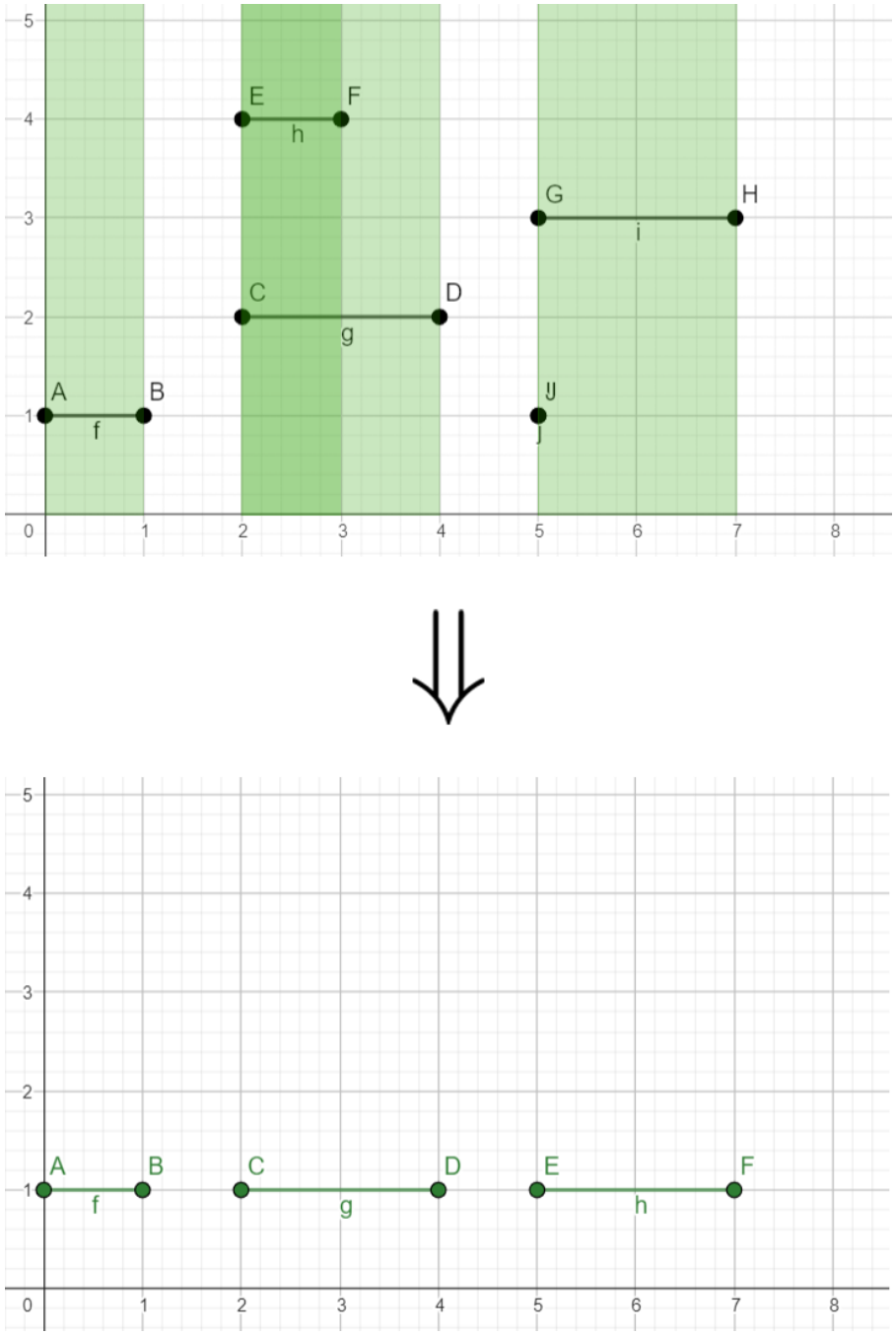


Рис. 22: Отрезки объединения

Между этими точками *cnt* мог быть и больше 0 - это означает что отрезки «накладывались», но точки, покрытые ими, все равно войдут в тот же отрезок объединения.

Код:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool comp(const pair<int, int> &f, const pair<int, int> &s){
    if (f.first == s.first) return f.second > s.second; //Если координаты точек равны, то
    //                                                  сравниваем по типу
    return f.first < s.first; //Иначе сравниваем по координате
}
signed main(){
    int n;
    vector<pair<int, int> > events;
    for (int i = 0; i < n; ++i) {
        int l, r;
        events.emplace_back(l, 1); //Открывающая отрезок
        events.emplace_back(r, -1); //Закрывающая отрезок
    }
    sort(events.begin(), events.end(), comp);
    vector<pair<int, int> > ans; //Отрезки объединения
    int last = -1, cnt = 0; //Изначально никакие отрезки не открыты
    for (auto &[x, type] : events) {
        if (type == 1) { //Отрезок начался
            if (cnt == 0) last = x;
        }
        else{ //Отрезок закончился
            if (cnt + type == 0) ans.emplace_back(last, x);
        }
        cnt += type;
    }
    return 0;
}
```

Асимптотика - $O(n \log n)$ (сортировка). Также обратите внимание на специальный компаратор.

II. Следующая задача - https://codeforces.com/gym/104119/problem/4.

Она встретилась на муниципальном этапе ВСОШ в Москве в 2022 году. Перед тем, как приступить к решению, обратим внимание на некоторые свойства в задаче:

1) Все координаты различные, следовательно в одной точке могут встретиться **не более** 2**-х** кораблей.

2) Все координаты чётные, следовательно расстояние между каждой парой кораблей **всегда** остается **четным**.

3) В **один и тот же момент времени** можно выпускать **несколько** торпед.

4) В **одну и ту же точку** можно выпускать **несколько** торпед в разные моменты времени.

Теперь мы можем представить корабли в задаче, как точки, двигающиеся вправо/влево и предъявить оптимальную стратегию по уничтожению кораблей.

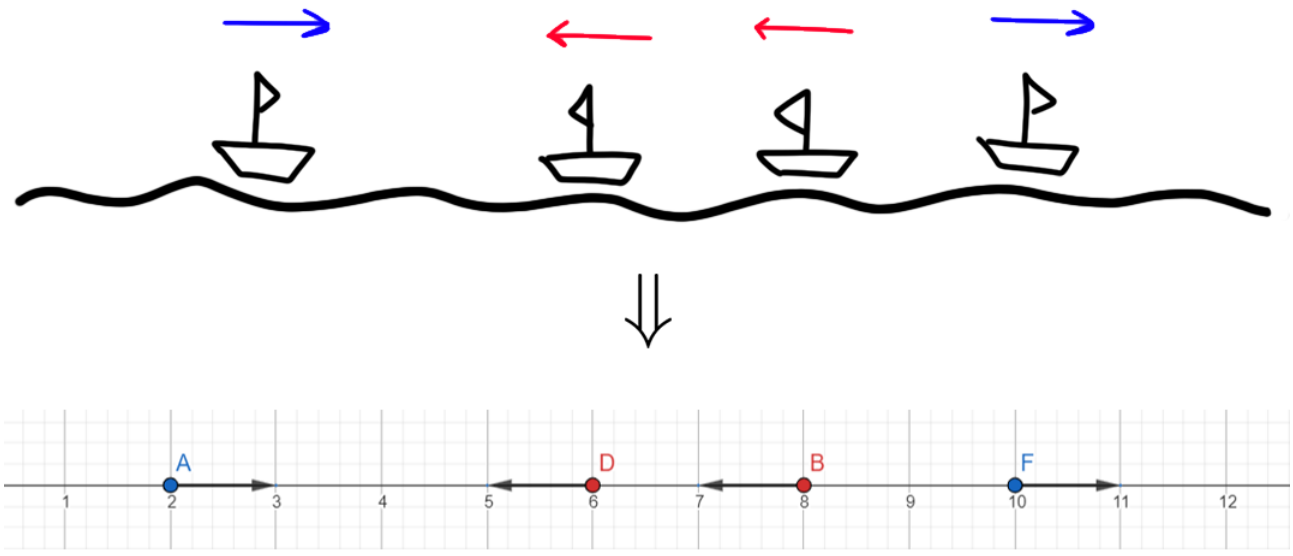


Рис. 23: Представление кораблей в виде точек

Считаем корабли обоих видов в один массив t и представим их в виде пар $(x_i, type_i)$, где x_i - координата корабля и $type_i$ - направление движения (1 - вправо, -1 - влево).

Отсортируем корабли сначала по координате, затем по направлению (корабли, плывущие вправо должны идти **раньше**, чем плывущие влево). Теперь мы сможем обрабатывать встречи, пройдя по массиву t слева направо.

Если мы встретили корабль, двигающийся вправо, положим его в отдельный стек. Если мы встретили корабль, двигающийся влево, то он обязательно встретится с кораблем на вершине стека (если стек не пустой).

По свойству (1), больше 2-х кораблей в этой точке **не будет**, а значит оптимально подбить обе цели в момент встречи, кинув 1 торпеду.

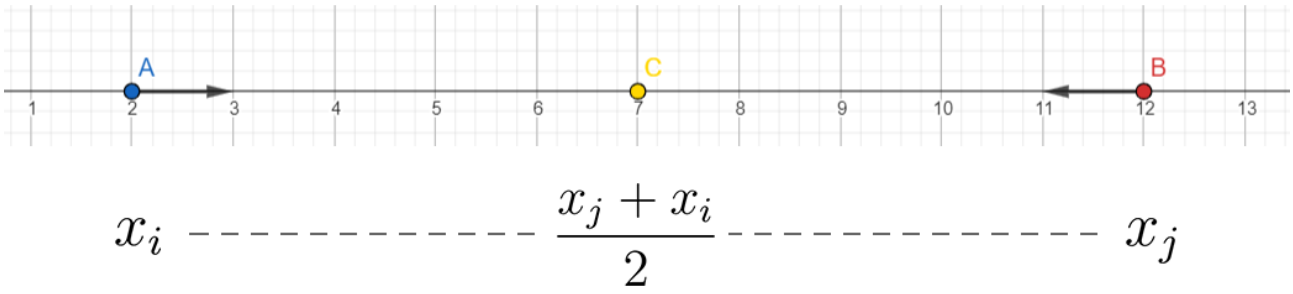


Рис. 24: Точка встречи двух кораблей

Два корабля в точках x_i и x_j ($x_i < x_j$) встретятся в точке $c = \frac{x_j+x_i}{2}$ в момент времени $t = \frac{x_j-x_i}{2}$. Сохраним параметры текущей торпеды, удалим подбитый корабль из стека и продолжим алгоритм.

В конце отдельно обработаем оставшиеся корабли. Их всех можно уничтожить заранее, например в момент времени 0.

Код: https://pastebin.com/eijP4nW8

III. Задача - https://codeforces.com/gym/103471/problem/4.

Эта задача встретилась в 2021 году на муниципальном этапе ВСОШ в Липецке. Как и с предыдущей задачей, посмотрим на ограничения и свойства в задаче.

Для начала решим задачу без запросов. Необходимо уметь проверять, что массив b является подпоследовательностью массива a.

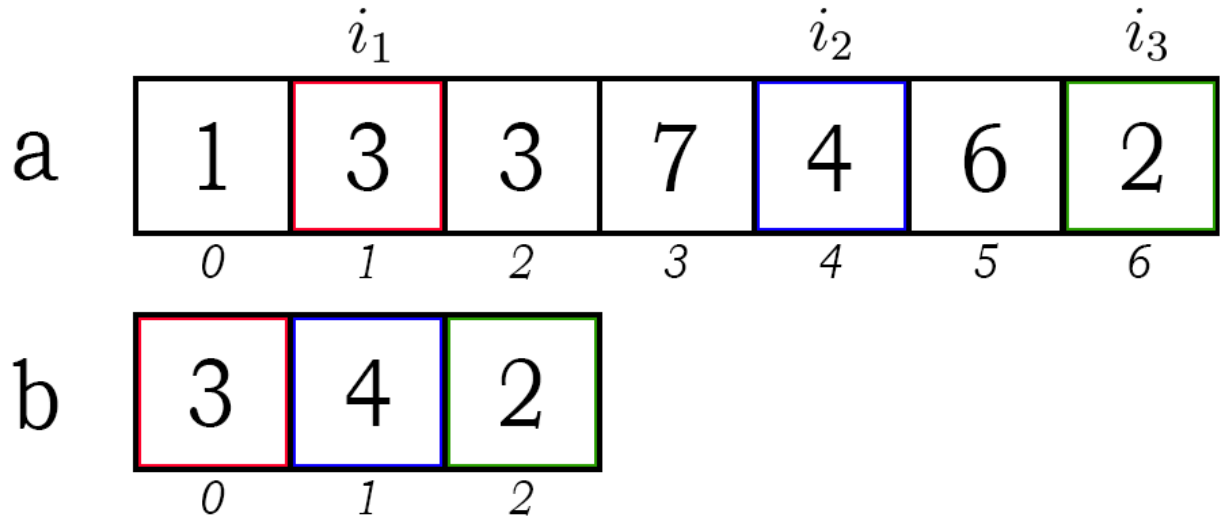


Рис. 25: Выделение подпоследовательности b в массиве a

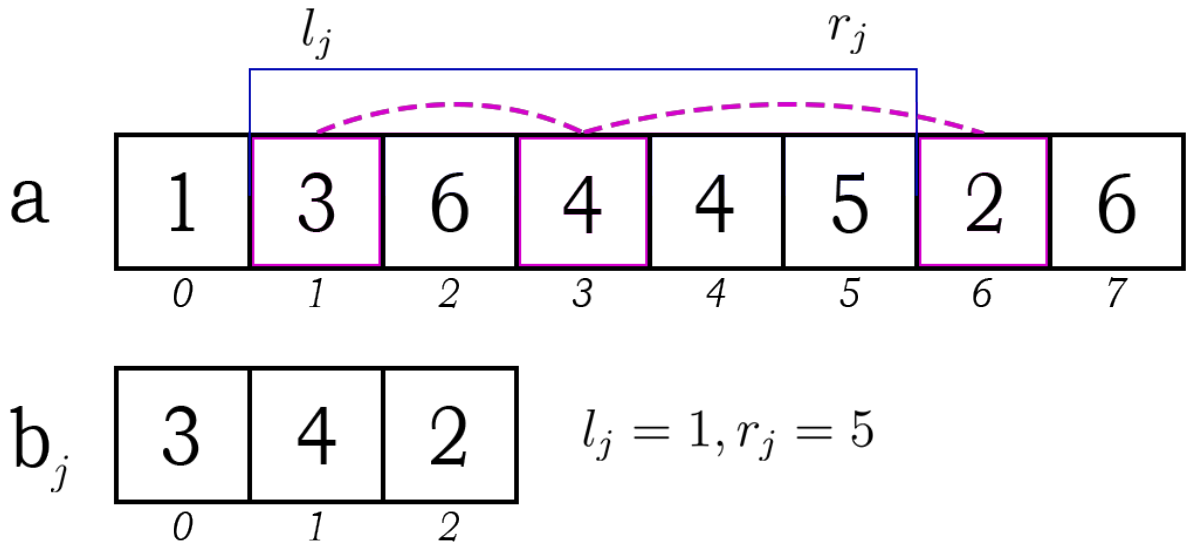
Формально, массив b является подпоследовательностью массива a, если существует набор из $|b|$ индексов $i_1 < i_2 < \ldots < i_{|b|}$ такой, что для каждого k выполняется $a_{i_k} = b_k$ (элемент на позиции i_k в массиве a равен элементу на позиции k в массиве b).

Пройдемся по массиву a и будем хранить текущий указатель на элемент в массиве b (изначально он указывает на первый элемент). Если мы встретили элемент, равный текущему в массиве b, то начинаем искать следующий из массива b в массиве a. Для этого просто увеличим указатель на 1.

Так как нам не нужно хранить позиции, в конце проверим, что все элементы нашлись (указатель указывает на конец массива b).

Если применить текущую идею, то мы решим задачу за $O(NQ)$ и получим 50/100 баллов. Проблема в том, что каждый запрос мы проходимся по массиву a от l_j до r_j. Ограничения на суммарный размер таких отрезков **нет** (как у массивов в запросах), поэтому наихудшее время работы достигается, если все Q запросов будут на отрезке $[1; n]$.

Идея для ответа на запросы останется, просто применим более «сжатый» подход к поиску элементов в массиве a.



Запомним позиции каждого элемента в массиве a. Так как $a_i \leq 10^6$, можно завести 10^6 векторов. Теперь, будем итерироваться не по массиву $a[l_j...r_j]$, а по массиву b из запроса и заведем указатель $last$ на последний найденный элемент на отрезке $[l_j; r_j]$ в a. Для каждого i мы бинарным поиском будем находить **ближайший справа** от $last$ элемент равный текущему и обновлять $last$. Чтобы найти ближайший равный, нужно в массиве $pos[b[i]]$ бинарным поиском найти позицию, **строго большую** $last$.

Таким образом, мы не итерируемся по ненужным элементам массива и сразу же рассматриваем $O(m)$ нужных нам позиций. Итого на запрос j мы отвечаем за $O(|b_j| * \log N)$. Значит асимптотика ответа на все запросы равна $O(\sum_{j=1}^{Q} |b_j| * \log n) = O((M+Q)\log N)$ и решение набирает 100/100 баллов.

Код: https://pastebin.com/vSBRDNqd

8.2 Практика

Дополнительные задачи:

- https://atcoder.jp/contests/abc355/tasks/abc355_d
- https://atcoder.jp/contests/abc328/tasks/abc328_d

9 Заключение

Линейные алгоритмы являются важной частью олимпиадного программирования. Их применение встречается в широком спектре задач, и зачастую они позволяют найти наиболее эффективные решения. Овладение этими алгоритмами способствует формированию прочной базы для более сложных техник.

Я надеюсь, что это пособие действительно помогло разобраться в теме, и подборка задач в конце каждой главы дала возможность не только закрепить знания, но и развить интуицию для более быстрого поиска решений.

Рекомендую пройти опрос, чтобы дать обратную связь по пособию, его наполнению и понятности пройденного материала:

https://forms.gle/EE8KQhPbcpYN4Gqg8

10 Список использованных ресурсов

10.1 Литература

1. Codeforces [Электронный ресурс] https://codeforces.com/ (Дата обращения 08.10.2024)
2. Алгоритмика [Электронный ресурс] https://ru.algorithmica.org/ (Дата обращения 08.10.2024)
3. Пособие peltorator'a [Электронный ресурс] https://peltorator.ru/cp_book.pdf (Дата обращения 08.10.2024)

10.2 Тестирующие системы

1. Codeforces https://codeforces.com/ (Дата обращения 08.10.2024)
2. Sort me https://sort-me.org/ (Дата обращения 08.10.2024)
3. Atcoder https://atcoder.jp/ (Дата обращения 08.10.2024)
4. Leetcode https://leetcode.com/ (Дата обращения 08.10.2024)